An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects

Antonio Terceiro, Luiz Romário Rios, Christina Chavez
Software Engineering Lab (LES)
Computer Science Department
Federal University of Bahia
{terceiro,luizromario,flach}@dcc.ufba.br

***Abstract*—Background: Several factors may impact the process of software maintenance and evolution of free software projects, including structural complexity and lack of control over its contributors. Structural complexity, an architectural concern, makes software projects more difficult to understand, and consequently more difficult to maintain and evolve. The contributors in a free software project exhibit different levels of participation in the project, and can be categorized as core and peripheral developers.**

Research aim: This research aims at characterising the changes made to the source code of 7 web server projects written in C with respect to the amount of structural complexity added or removed and the developer level of participation.

Method: We performed a observational study with historical data collected from the version control repositories of those projects, recording structural complexity information for each change as well as identifying each change as performed by a core or a peripheral developer.

Results and conclusions: We have found that core developers introduce less structural complexity than peripheral developers in general, and that in the case of complexity-reducing activities, core developers remove more structural complexity than peripheral developers. These results demonstrate the importance of having a stable and healthy core team to the sustainability of free software projects.

I. Introduction

The problem of software aging, introduced by Parnas in 1994, is a well-known problem in the Software Engineering field [1]. It has two causes: the first is "lack of movement", when a software project fails to deliver an updated product that fulfils the changing needs of its users. The second cause of software aging is "ignorant surgery": the software is subsequently changed by people who do not fully understand its design, and after some time even the creators of the design do not understand it anymore; changing the software becomes harder and more error prone.

According to Parnas, software aging is inevitable, just like human aging. While there are measures that can be taken to slow down the effects of software aging, even successful projects will get old and suffer the consequences of aging: older software requires more effort for being updated as time passes, bug fixes tend to cause new bugs, and can even exhibit unsatisfactory performance.

The aging phenomenon can be also observed in the context of free software[1] projects. For instance, GNOME is a worldwide project that provides the GNOME desktop environment, a desktop system, and the GNOME development platform, a framework for building applications that integrate into the rest of the desktop. GNOME had two of its components, `eog`, GNOME's image viewer and `gnome-session`, GNOME's session management software, completely rewritten from scratch.[2]

In older projects, the code base becomes so difficult to maintain that their maintainers decide that rewriting them will require less effort than to keep maintaining it. Sometimes the developers believe that rewriting their project (or large parts of it) is absolutely necessary for them to be able to evolve the project. Less complex code may facilitate the addition of new features and bug fixing in a sustainable way and therefore supports maintainability.

These rewrites take time and effort, so every project leader would certainly prefer not to deal with them. If the factors that contribute to added complexity in free software projects can be identified and characterized, project leaders would possibly be able to avoid such rewrites.

Several factors may impact the process of software maintenance and evolution. The internal quality of the software that is to be evolved and maintained is one of the most important: the lower the quality of the source code and other artifacts, the larger is the effort to change them. In this context, the *structural complexity* of the source code is one of the dimensions of the internal quality.

One of the characteristics of free software projects is the lack of control over the project developers. They usually join

[1]in our work, "free software" is used as a synonym for "open source software" (OSS), 'free/open source software" (FOSS) and "free/libre/open source software"(FLOSS).

[2]These rewrites were described in their wiki pages, respectively http://live.gnome.org/EyeOfGnome/EogNg and http://live.gnome.org/SessionManagement/NewGnomeSession.

and leave projects based on their motivation or needs, and present different *levels of participation*. The convention is to categorize them as either *core developers* or *peripheral developers*. The former are the most active developers, who perform most of the work, and are in general in charge of the important decisions regarding the project. The later contribute less often, and normally have little decision power in the project [2], [3].

In this paper, we explore the different levels of developers participation as a factor that may influence the amount of structural complexity in free software projects. We want to verify whether core and peripheral developers introduce different amounts of structural complexity in the code, and also whether they remove different amounts of complexity during activities that reduce the complexity of the source code, such as refactorings.

The remainder of this paper is organized as follows: section II presents the background for this study; section III presents the hypotheses that we investigate in this paper; section IV describes our research design; in section V we analyse the results obtained; section VI discusses threats to the validity of this study; section VII discusses similar studies and compares our study with them; finally, in section VIII we discuss our results as well as possibilities of future work.

II. Background

The following sections present background in the main subjects addressed by this paper: free software projects, structural complexity of software and the concept of Core and Periphery as different levels of contribution in free software projects.

A. Free Software

We call "free software" every software product that is available to its users under a license that allows it to be freely[3] studied, modified, and redistributed, according to the "Free Software Definition" [4], published by the Free Software Foundation.

The most interesting aspect of the nature of free software, from a Software Engineering point of view, is its development process. A free software project starts when an individual developer, or an organization, decides to make a software publicly available on the internet so that it can be freely used, modified and distributed. After an initial version is released, and with some effort in advertising it in the appropriate channels, independent users start using it, and those users who are also software developers are able to inspect the software's source code and propose changes to it. These changes are sent back to the original developer(s) in the form of *patches*[4]. The project leader(s) review this change proposals and apply (or not) them to their own version, so that when a new version of the software is released, end users will have access to the new functionalities of bug fixes proposed by these contributors.

[3]As free software advocates are used to remind, the "free" here is "free" as in "freedom", not as in "free lunch" ("gratis").

[4]A patch is a file that describes the changes to one or more files, normally of textual content, by describing which lines to remove and which lines to add. After receiving a patch the developer can reproduce the changes proposed by the sender in his/her own copy of the source code.

In the course of time, the most frequent contributors gain the trust from the initial developer(s), and can receive direct write access to the official source code of the project. That way they will be able to make changes directly instead of having to pass through the review of the original developers. This process of developers joining free software projects may range from very informal to very formal, depending on the project. Small projects normally have informal procedures for that, e.g. one existing developer just offers an account in the version control system to the new contributor. Larger projects, on the other hand, may have more "bureaucratic" processes for accepting a new developer with privileged access to the project's resources, such as filling forms with an application, signing terms of copyright transference and other procedures.

The development is normally driven with the help of a version control system (VCS), where the latest version of the source code is stored in a source code repository. The VCS records every change ever made to the source code, as well as the author and date of the changes. While the repository is often publicly available for read access, write access is restricted to a limited group of developers. Other developers will need their changes (*patches*) to be reviewed by a developer with the needed privileges in order to get their contributions into the project's official repository. This process is illustrated by figure 1.

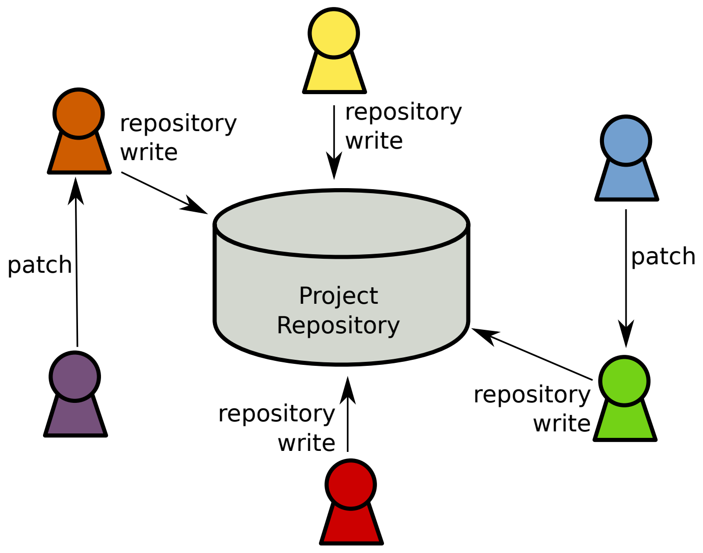

Fig. 1. Free software development by means of a VCS repository.

There are variants of this process. For example, recently the usage of distributed version control became quite popular. With distributed VCS, there is no need for a central repository. Each developer has its own repository, which may be published for other to access. But the concept of an official repository, blessed by the project leaders, still exist.

The following characteristics make free software projects different enough from "conventional" software projects, and also makes them an interesting object of study:

- **Source code availability.** Source code of free software projects is always available on the internet. Although most of the projects have a publicly-accessible version control repository, there are projects that do not have one (e.g. the Lua Programming language [5]).
- **User/developer symbiosis.** In most free software projects the developers are also users of the software, and they also provide requirements. Maybe because of that, several free software projects do not have explicit requirement documents, and the development flows on a rhythm in which the developers are able to satisfy their own needs.
- **Non-contractual work.** A large amount of work in free software projects is done in a non-contractual fashion. This does not imply that the developers are necessarily volunteers, but only that there is no central management with control over all of the developers' activities.
- **Work is self-assigned.** Since most free software projects don't have a central central management with control over the contributors' activities, the work of these contributors is normally self-assigned: volunteer developers work on the parts of the project that most appeal to them.
- **Geographical Distribution.** In most Free Software projects the developers are spread among several different locations in the world. In the projects with high geographical dispersion, almost all communication is performed through electronic means.

Although the Software Engineering literature tends to portrait Free Software as a homogeneous phenomenon[6], most of these characteristics do not apply to all free software projects, and some of them may be manifest in different ways across projects.

B. Structural complexity

Structural complexity is an architectural concern: it involves both the internal organization of software modules, as well as how these modules relate to each other [7], [8]. structural complexity influences the developer's time: a more complex software is expected to require more effort from developers to be comprehended in maintenance activities [9].

Several aspects of Software Design can be considered when evaluating structural complexity. We can consider, among others, coupling [10], [11], cohesion [10], and inheritance [10], [11].

While inheritance is specific to the object-oriented paradigm, coupling and cohesion are more generally applicable. Every programming paradigm has a notion of *module*, whether it is called "module", "class", "aspect", "abstract data type", "source file", etc. Having modules, one can always analyse a program and identify which other modules a module refers to and thus have a notion of coupling, and also verify how the subparts of a given module interact with each other to evaluate the cohesion of such a module.

In an experimental setting with professional software developers, Darcy *et al* found that more complex software requires more effort for maintenance activities [9]. Moreover, they verified that neither coupling nor lack of cohesion by themselves could explain the decrease in comprehension performance of the developers; only when considered together (by multiplying the two) they presented an association with higher maintenance effort. The authors claim that when considering structural complexity, one must consider coupling and cohesion together.

In this work, we follow Darcy *et al* and consider both coupling and cohesion in our definition of structural complexity. We provide, however, a formal definition of this metric, as follows.

The structural complexity of a module is given by multiplying coupling (Chidamber and Kemerer's CBO [10]) and lack of cohesion (Hitz and Montazeri's LCOM4 [12]) metrics. A project-wide value for structural complexity can be obtained by taking the average structural complexity among all modules. Given $M(p)$, the set of modules in a project p, we have:

$$SC(p) = \frac{\sum_{m \in M(p)} CBO(m) \times LCOM4(m)}{|M(p)|}$$

C. Structural Complexity in free software projects

Midha [13] studied projects from sourceforge.net and verified that increases in complexity leads to increase in the number of bugs in the source code, decrease in contributions from new developers and increase in the time taken to fix bugs. Although using a different concept of structural complexity by considering MacCabe's Cyclomatic Complexity and Halstead's Effort[5], these results demonstrate that complexity has nasty effects on Free Software projects. We speculate that an increase in structural complexity as defined in this study here has similar effects in free software projects, as it did in the controlled experiment by Darcy *et al* [9].

Stewart *et al* studied 59 projects written in Java that were available on Sourceforge [14], using the product of coupling and lack of cohesion as their structural complexity measure. They verified 4 different patterns of structural complexity evolution, of which 2 presented growing trend in the end of the period. The other 2 presented stabilization in the end of the period: none of the identified patterns featured a complexity reduction trend. A previous study of ours also indicated a growing trend in structural complexity on another (but smaller) project written in C [15].

Increasing complexity trends are not an exclusive feature of free software projects, and are not recent news: the seminal work of Lehman on software evolution already identified it, and that led to the formulation of the second law of software evolution the Law of Increasing Complexity [16]. That law, formulated in the context of studies on proprietary software systems, states that as systems evolve, their complexity increases unless work is done to maintain or reduce it.

[5]These two measures represent respectively the internal complexity of subroutines and the overall vocabulary size of the code base. They reflect a different aspect of structural complexity, at the subroutine level. Here we are looking at structural complexity at the design/architecture level, considering the relationship between modules and between the sub-parts of each module.

For now, we know that i) software complexity is associated with undesirable effects (more maintenance effort, more bugs, less attraction of new developers) and ii) structural complexity tends to not decrease, and in a reasonably large amount of cases, it tends to grow. That leads us to the following question: why does structural complexity increase in the context of Free Software projects?

In this paper we investigate whether developer attributes can explain the variation in structural complexity in free software projects, specifically whether core and periphery developers contribute differently to such variation. The concept of core and periphery an important aspect in the study of the free software development process, and is described in the next section.

D. Core and Periphery in free software Projects

Normally, a Free Software project is started by a single developer, or by a group of developers, in need of addressing a particular need. After there is a usable version, it is released to the public under a Free Software license which allows anyone to use, change and distribute a copy of that software. As new users get interested in the project, some of them may start to contribute to it in several possible ways: with code for new features or bug fixes, with translations into their native languages, with documentation, or with other types of contribution. At some point, then, the project has a vivid and active community: a group of people that gravitate around a project, with varied levels of involvement and contribution.

The "onion model" [2], [3] became a widely accepted representation of what happens in a Free Software project, by indicating the existence of concentric levels of contribution: a small group of core developers do the largest part of the work; a larger group makes direct, but less frequent contributions in the form of bug fixes, patches, documentation, etc; an even larger group reports problems as they use the software, and the largest group is formed by the silent users who only use the software but never provide any type of feedback.

The processes by which participants migrate from one group to another are very different from one community to the other: communities may adopt more formal and explicit procedures for that, or use a more relaxed approach and let things flow "naturally". But in general the achievement of central roles (and thus more responsibility, respect and decision power) are merit-based: a developer becomes a leader by means of continuous valuable contributions to the community [17].

Since most of the work is done by a core team, it is important for projects to keep a healthy and active core team. Some projects are able to keep its core team with few or no changes across its entire history, while others experience a succession of different generations of core developers [18], [19].

The relationship between core contributors and peripheral (non-core) members of a community are not always smooth: sometimes the core tends to work on their own demands and to give little attention or even to ignore completely the demands of the periphery [20], [21]. From an individual point of view, core and periphery members also exhibit different behaviour while debating subjects related to the project [22] or in the bug reporting activity [21].

III. Research hypotheses

As discussed earlier, structural complexity raises the maintenance cost of a software project, because the code becomes harder to understand, and in consequence harder to modify. In free software projects, such an increase in effort may represent an extra difficulty for gathering new contributors. Failing to attract contributors represents a threat to the project sustainability [23], specially those which are not mainly funded by a single organization and rely on the contributions of volunteers.

There are differences between core and peripheral contributors with respect to the volume of work done and behaviour in communication inside the project. This begs the question as if the quality of their contributions could be also different. We want to evaluate the amount of complexity that they introduce into the code. Since core developers have deeper knowledge of the software architecture, it is expected that their changes to the source code do not add as much structural complexity as the changes made by peripheral developers do. Thus the first hypothesis we want to test in this paper is the following:

> H_1: *changes made by core developers introduce less structural complexity than those made by periphery developers*.

Several projects also undertake development effort in order to refactor the code and thus reduce complexity [24]. As formulated by Lehman [16], this is needed in order to keep the project under a sustainable level of complexity. While both core and periphery developers can participate in such a task, core developers are probably more successful at it than periphery developers, and we want to verify that empirically. Our second hypothesis is then related to the reduction of structural complexity in the source code:

> H_2: *among the changes that reduce structural complexity, the ones made by core developers achieve greater structural complexity reduction than those made by periphery developers*.

IV. Research Design

In order to test our hypotheses, we designed and executed an empirical study, which is described in this section. Care was taken in order to provide all information needed by the reader to assess the quality of the study and the applicability of its results [25].

The research method used was an *observational study*, in which a phenomenon is observed in its natural setting (as opposed to a controlled lab environment, used in a true experiment).

Our data collection approach was to mine the version control systems of a selected group of free software projects from the web server application domain, and to collect data from each change to the project source code. For each change, we registered the date of the change, the variation of structural

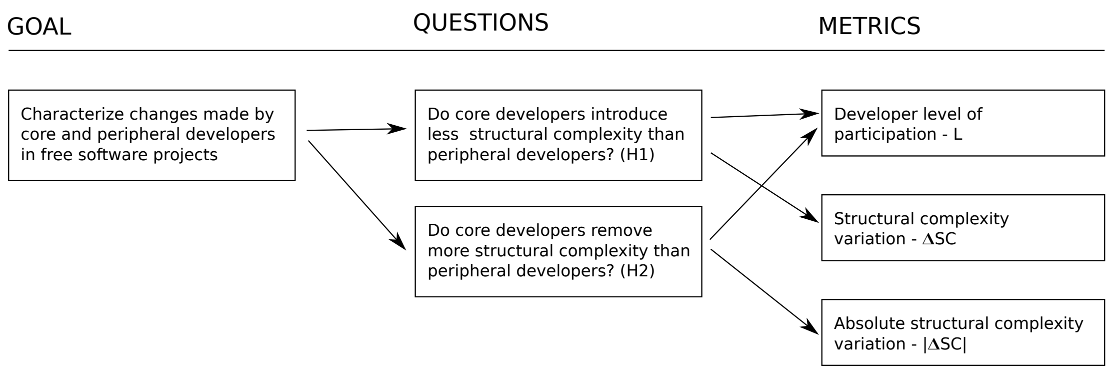
GOAL
QUESTIONS
METRICS
Characterize changes made by core and peripheral developers in free software projects
Do core developers introduce less structural complexity than peripheral developers? (H1)
Do core developers remove more structural complexity than peripheral developers? (H2)
Developer level of participation - L
Structural complexity variation - ΔSC
Absolute structural complexity variation - |ΔSC|

Fig. 2. GQM diagram of the study

complexity accomplished by the change, and whether the change was made by a core or a peripheral developer.

The study definition, using a GQM template [26] is as follows: In this study we **analyse** changes made to the source code of free software projects **for the purpose of** of characterization **with respect to** structural complexity added or removed and level of developer participation, **from the perspective** of the researcher **in the context of** the web server application domain.

Figure 2 presents a GQM diagram of the study. In such diagram, the first column, "Goals", identifies the problems that a given study is trying to solve. "Questions" identifies the questions that, when answered, will provide a solution to the problems, and "Metrics" identifies which metrics (or "variables") need to be measured so that we achieve answers to the related questions. Arrows connect goals and their associated questions, as well as questions and their associated metrics. In the case of the current study, there is a single goal with two questions. Our questions are directly mapped to the research hypotheses presented in section III.

The metrics associated to each research question in figure 2 are presented in section IV-A. Our data collection approach and the obtained sample are described in section IV-B.

A. Variables and operational definitions

The following variables are considered in the study:

- Independent variable
  - L, the level of involvement of the author of the given change in the project at that point in time. This is determined by splitting the entire studied period in 20 periods of equal duration, and for each period identifying the 20% top committers as the core team (cf. [18], [19]). The reader should note that by according to this definition, the same developer can be considered as a core developer in some periods and a peripheral developer in other periods. This is coherent with reality: since developers often do not have any formal responsibilities with with the projects, they may reduce or increase their activity in the project in specific periods, and thus can shift from the core to the periphery and vice-versa.
- Dependent variables
  - SC, the overall structural complexity of the project after each change, as described in section II-B.
  - ΔSC, the increment in structural complexity in each change. For each change, this value is obtained by subtracting its SC value from the previous change's SC value. This variable represents how much the structural complexity changed after a given change was applied to the project source code.
  - $|\Delta SC|$, the absolute change in structural complexity (i.e. the absolute value of ΔSC).

B. Sample and data collection

We started by identifying all web server packages in the Debian GNU/Linux archive. This decision was made in order to select projects that were being actually used: if a web server software is properly packaged and maintained in Debian, it means that there is interest in it to the point that someone volunteers to maintain an automatic installation package so that users can install it without the need for manual configuration[6].

This provided us 21 web server projects as a starting point. Since we needed to harvest the version control history of the projects in order to characterize the changes made to the source code, we needed that the projects have an accessible version control repository. Due to a temporary limitation in our tooling for source code analysis, we had to stick to projects written in the C language. Since C is a language commonly used for infrastructure software, most of the web servers identified are written with it. After applying both restrictions, we could identify 7 projects to work with. They are listed in table I.

The source code repository of each project was imported locally in a `git`[7] repository in order to facilitate fast and offline history browsing. We used a set of scripts developed by us to mine this repository as follows:

- Determine the list of relevant commits, by identifying the commits that changed source code files. This way we

[6]Although the first author of this paper is a Debian maintainer, he is not maintainer of any of the web server packages evaluated. Also, none of the authors is affiliated with any of the studied projects.

[7]http://git-scm.org/. `git` has support for importing repositories from CVS and Subversion.

TABLE I
PROJECTS SELECTED FOR ANALYSIS

Project	Start	End	Commits	Developers
aolserver	2000/05	2009/05	1125	22
apache	1999/06	2009/11	9663	72
cherokee	2005/03	2009/10	1545	8
fnord	2001/08	2007/11	99	2
lighttpd	2005/02	2009/10	775	6
monkeyd	2008/01	2009/06	207	4
weborf	2008/07	2009/10	139	3

avoided analysing subsequent states of the source code that were no different from each other.

- Checkout each relevant version and run a static source code analysis tool in order to calculate the source code metrics used, namely CBO [10] and LCOM [10] (we used the improved version from Hitz and Montazeri [12], though).
- Accumulate the results for each change in a single data file per project.

After processing all the projects, we loaded their raw data in a relational database in order to facilitate the evaluation of L, SC, ΔSC and $|\Delta SC|$. During this process we excluded 7 commits that had no previous commit to compare to (i.e. the very first commit of each project).

Table I lists some aggregated information about the data sample. The "Start" and "End" columns show year and month of the first and last changes considered, respectively, according to the project's version control system. The "Commits" column displays the number of changes (commits) considered for each project, and the "Developers" column counts how many different developers were responsible for these considered changes.

The following tools were used to mine the repositories:

- The static source code analysis was performed with `analizo`[8], a multi-language source code analysis tool we have been working on.
- The history analysis of the version control repositories was made by the `analizo-metrics-history` script from the `analizo-utils` package[9]
- The analysis specific to this study was done by some ad-hoc Ruby code.

The complete package for this study, with data, local scripts etc is available on the web at the following address: http://www.dcc.ufba.br/~terceiro/papers/cpsc.tar.gz.

V. Data Analysis and Results

The full dataset contained 13553 changes, of which 9944 (73.36%) were made by core developers, and 3609 (26.63%) by peripheral developers. In order to test our hypotheses, we excluded from the analysis the changes that did not make any changes to the structural complexity metric SC (i.e. $\Delta SC = 0$). The remaining changes are similarly distributed between the two groups: of 2513 changes, 1994 (79.35%) were made by developers considered as core developers, while the other 519 (20.65%) were made by peripheral developers.

To test H_1, we need to compare ΔSC for the subset of changes made by core developers and the ones made by periphery developers. H_1 can then be formalized as follows:

$$H_1 : \mu_{\Delta SC_{core}} < \mu_{\Delta SC_{periphery}}$$

We used a t-test to verify the hypothesis, and were able to reject the null hypothesis that there is no difference between the means and accept the alternative hypothesis H_1, with $p < 0.05$ ($p = 0.01515265$). This demonstrates that our hypothesis that *changes made by core developers introduce less structural complexity than those made by periphery developers* is supported by the data.

To test H_2, we must consider only the cases in which there is a decrease in structural complexity. To do that, we filtered the dataset again and kept only the changes in which $\Delta SC < 0$. In this filtered dataset we have 1165 changes, of which 939 (80.60%) were made by core developers and 226 (19.40%) by periphery developers. We want then to verify whether the amount of structural complexity removed by core developers is greater than the amount removed by periphery developers, what can be formalized as follows:

$$H_2 : \mu_{|\Delta SC|_{core}} > \mu_{|\Delta SC|_{periphery}}$$

The t-test for H_2 allowed us to reject the null hypothesis of the two variables being equal, and accept the alternative H_2 with $p < 0.05$ ($p = 0.01091324$). The data support our second hypothesis as well: *among the changes that reduce structural complexity, the ones made by core developers achieve greater structural complexity reduction than those made by periphery developers.*

Table II presents descriptive statistics of the dataset used in these results.

The data analysis was performed with the R system [27] and RKWard, a frontend to R[10].

VI. Threats to Validity

While carefully designed, this study has some limitations that represent threats to its generalisability.

The careful reader will notice that although all variables tested in section V are not normally distributed (see table II), we still used the t-test for comparing them. The t-test usually requires that the variables have a normal distribution, but as noted by Wohlin *et al* [28], it is robust enough to support some deviation from these preconditions. In particular, since our sample is large enough, we can use the t-test without problems. To be sure, we also performed a Wilcoxon/Mann-Whitney test (a non-parametric test indicated as replacement for the t-test when the samples are not normally distributed) that provided similar results.

With respect to the choice of data sample, by considering only one application domain and only projects written in C we do not address the wide diversity of free software projects.

[8]http://github.com/terceiro/analizo
[9]http://github.com/terceiro/analizo-utils

[10]http://rkward.sourceforge.net/

TABLE II
DESCRIPTIVE STATISTICS OF THE VARIABLES TESTED

Variable	Mean	Std. Dev.	Min.	Max.	n
ΔSC_{core}	0.001660474	0.3334254	-5.967357	5.355073	1994
$\Delta SC_{periphery}$	0.03426117	0.2970714	-2.023467	3.021991	519
$\|\Delta SC_{core}\|$	0.1291047	0.3092991	5.939609e-05	5.967357	939
$\|\Delta SC_{periphery}\|$	0.09200304	0.1891808	0.002171662	2.023467	226

In order to have results that can be properly generalized, we need to study a more diverse population. It may be the case that the communities working on different application domains or different programming languages have different design and programming practices, and that could affect the obtained results.

From a construction validity point of view, by having a single independent variable (the level of involvement of the developer) we are not considering other factors that influence the addition of structural complexity to the source code on those projects.

A limitation caused by our choice to use only the version control metadata directly provided by the repositories in structured form is that we may be masking the reality by considering the commit author as being the same developer who actually developed the change. In several projects there is a limited set of developers, known as *committers*, who have write access to the repositories. This way, contributions from developers who are not committers need to be reviewed and applied by a committer, and the version control repository stores the committer name as the author of the change. In those cases the committer usually gives credit to the original author of the change in free-form text inside the commit log message, but we did not consider this in the data extraction. On the other hand, in such cases the committer explicitly decided to approve a change proposed by another developer and to apply that change to the source code; one can also argue that by doing that the committer took part in a design decision that affects the structural complexity of the project, even not having written the code herself.

We did not analyse the nature of the changes that reduce structural complexity, so we cannot claim that they are similar. Complexity-reducing changes may actually represent corrective, adaptive, perfective and preventive maintenance activities. Those changes can also be localized in few software modules, or systemic changes that touch a large number of modules. They can be defect corrections, or implementation of new features. For example, a change that adds a new module will usually make the structural complexity metric drop: since we used the average structural complexity per module, adding a new module that is not as complex as the current average will make the average value fall down. In such case, however, adding a new module that is not as complex as the average may be considered a good thing, since we are adding new functionality to the software without making the part that implements that functionality as complex as the rest of the system. The bottom line is that the differences in structural complexity reduction may actually be caused by the type of maintenance being performed while core and periphery developers happen to perform different types of maintenance.

We also excluded from the analysis changes that do not change the structural complexity metric (i.e. those in which $\Delta SC = 0$). These changes may also reveal interesting design activities, such as removing a dependency from module A to module B and making A depend on C instead. While this will not change A's coupling, being able to analyse changes like this one may provide relevant information about the project's design activity. By not considering this type of change we are probably ignoring events that may influence the others that actually change the structural complexity metric.

VII. RELATED WORK

Capra *et al* [24] claim that free software projects with a more open governance structure exhibit a better design quality. From one side, a higher design quality enables a more open governance: less coupled modules allow different developers to work on their own parts of the project without explicit coordination activities. Having a more open governance gives developers more freedom to enhance the design quality instead of having to keep up with deadlines or another types of pressures from higher management or customers.

Bargallo *et al* found that project popularity[11] may be associated with a lower design quality. They argue that as a project becomes more popular, their lead developers may redirect their efforts from programming to other activities such as answering users in forums or mailing lists, reviewing contributions etc. Such projects, having their main (and more experienced) developers change their attention to non-programming activities, may suffer from a decrease in design quality[29].

These works try to find factors that influence what they call "good design" in free software projects, in terms of different sets of object-oriented metrics from the suites by Chidamber amd Kemerer [10] and Brito e Abreu [11]. Our work differs from the above by i) considering the structural complexity construct, which is based on concepts applicable to both OO and non-OO software (coupling and cohesion) and ii) studying developers' attributes as factors (as opposed to looking at organizational and project-wide aspects).

VIII. CONCLUSIONS

In this paper we have investigated the relationship between the different levels of developer involvement in free software projects and the amount of structural complexity added by changes recorded in version control systems. We thus provide

[11]measured as a function of the number of downloads, web traffic and development activity

relevant results on the technical side of the core/periphery dichotomy: core and periphery do not only behave differently [21], [22] and contribute different amounts of effort [3], [2], but they also provide code of different complexity.

The data obtained supports both our hypothesis: the core developers are able to make changes to the source code without introducing as much structural complexity as the peripheral developers; and they also remove more structural complexity than the other developers. Finding empirical evidence for such hypotheses highlights the importance of a healthy core team for a free software project: in a certain sense, the core team is responsible for keeping the project's conceptual integrity, as suggested by Brooks [30].

It is important to note, however, that these results cannot be taken as an incentive to not accept contributions from non-core developers. Not all projects are able to keep the same core team during its entire lifespan [18], so receiving new contributors is fundamentally important for the project's sustainability. In several projects, non-core developers are responsible for a healthy ecosystem of extensions, plugins and other types of add-ons that can be used together with the main product. The source code for these add-ons sometimes does not even reside in the main project repository, and sometimes the changes they propose to the core product are needed to enable an entire new line of possibilities for extension developers.

Attracting as much contributors as possible is important to have a thriving free software project. Project leaders will most likely not want to send their contributors away even if their contributions are not perfect. The results presented here can be seen as a opportunity for project leaders to qualify their contributors: since non-core developers tend to produce more complex code, it is perhaps a good idea to explicit review their code. If code review practices are adopted, core developers can work together with non-core developers looking for less complex solutions. This code review activity can even leverage documented guidelines of good design practices for the project contributors.

The work reported in this paper is part of a larger research project, in which we investigate how developer characteristics influence the variation of structural complexity in free software projects. Considering the developer's level of participation is just the first step towards a more comprehensive understanding of the phenomenon. That said, in the following paragraphs we outline directions that may be taken for future work.

We plan to test other developer attributes in order to build a more comprehensive model of how the developers influence the variation of structural complexity in these projects. Such attributes can be divided in two groups: i) attributes related to the developers themselves, such as skill level, programming ability, experience in general; ii) attributes related to the participation of the developer in the project, such as level of participation (already explored in this paper), experience in the project, experience with the modules being changed, and whether the developer is a specialist or not in the area he/she is working on (with regard to its own past activity in the project).

Another alternative is testing the same hypotheses tested in this study on each project individually. This may provide us with stronger results for specific projects, and for other projects their data may not support the hypotheses at all. If that is the case, doing a richer characterization of the projects would support us on identifying other factors that enable or prevent the results achieved in this study. We can also investigate other factors that may impact the introduction of structural complexity in the source code by constructing a richer characterization of the changes themselves.

Extending our dataset to include projects from other application domains and written in different programming languages will also help us generalize the results to a wider range of free software projects.

The information stored in the version control repositories can also be explored better. There is a lot of possibilities for analysing the combination of the metadata from the version control system *and* information about the actual changes in the structure of the source code (as opposed to mere information on lines added/removed).

It is also unclear, and worth investigating, how the developer's design ability advances as the developer advances in the community. It would be interesting to learn how individual developers evolve in terms of complexity added to the source code as a developer moves from the periphery to the core, or the other way around.

Acknowledgments

The authors are thankful for the insightful contributions from Dr. Daniela Cruzes from the Norwegian University of Science and Technology[12] and from their colleagues at the Software Engineering Lab (LES) at Federal University of Bahia[13], in special Dr. Manoel Mendonça. Antonio Terceiro is supported by the Brazilian National Council of Research and Development (CNPq)[14]. Luiz Romário is supported by *Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB)*[15]. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES)[16], funded by CNPq, grant 573964/2008-4.

We are also grateful to the anonymous reviewers for their comments and contributions. All of their remarks that could not be addressed in this paper will certainly be taken into account in future studies.

References

[1] D. L. Parnas, "Software aging," in *ICSE '94: Proceedings of the 16th international conference on Software engineering.* Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 279–287.

[2] K. Crowston and J. Howison, "The Social Structure of Free and Open Source Software Development," *First Monday,* vol. 10, no. 2, 2005.

[3] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 11, no. 3, pp. 309–346, 2002.

[12] http://www.idi.ntnu.no/

[13] http://les.dcc.ufba.br/

[14] http://www.cnpq.br/

[15] http://www.fapesb.ba.gov.br/

[16] http://www.ines.org.br/

[4] Free Software Foundation, "The Free Software Definition," 2009, Available at http://www.gnu.org/philosophy/free-sw.html, last accessed on January 1st, 2010.

[5] The Lua Programming Language, "Frequently Asked Questions," 2009, Available at http://www.lua.org/faq.html#1.8, last access on January 5th, 2010.

[6] T. Østerlie and L. Jaccheri, "A Critical Review of Software Engineering Research on Open Source Software Development," in *Proceedings of the 2nd AIS SIGSAND European Symposium on Systems Analysis and Design*, W. Stanislaw, Ed. Gdansk University Press, 2007, pp. 12–20.

[7] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architecture : Views and Beyond*, ser. The SEI series in software engineering. Boston: Addison-Wesley, 2002.

[8] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[9] D. P. Darcy, C. F. Kemerer, S. A. Slaughter, and J. E. Tomayko, "The Structural Complexity of Software: An Experimental Test," *IEEE Transactions on Software Engineering*, vol. 31, no. 11, pp. 982–995, Nov. 2005.

[10] S. Chidamber and C. Kemerer, "A metrics Suite for Object Oriented Design," *IEEE Trans. Sftware Eng.*, vol. 20, no. 8, pp. 476–493, 1994.

[11] F. B. e Abreu, "The MOOD Metrics Set," in *Proc. ECOOP Workshop Metrics*, 1995.

[12] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems," in *Proceedings of the International. Symposium on Applied Corporate Computing*, 1995.

[13] V. Midha, "Does Complexity Matter? The Impact of Change in Structural Complexity on Software Maintenance and New Developers' Contributions in Open Source Software," in *ICIS 2008 Proceedings*, 2008.

[14] K. J. Stewart, D. P. Darcy, and S. L. Daniel, "Opportunities and Challenges Applying Functional Data Analysis to the Study of Open Source Software Evolution," *Statistical Science*, vol. 21, p. 167, 2006.

[15] A. Terceiro and C. Chavez, "Structural Complexity Evolution in Free Software Projects: A Case Study," in *QACOS-OSSPL 2009: Proceedings of the Joint Workshop on Quality and Architectural Concerns in Open Source Software (QACOS) and Open Source Software and Product Lines (OSSPL)*, M. Ali Babar, B. Lundell, and F. van der Linden, Eds., 2009.

[16] M. M. Lehman, J. F. Ramil, P. D. Wernick, and D. E. Perry, "Metrics and Laws of Software Evolution-The Nineties View," in *Proceedings of the 4th International Symposium on Software Metrics*, 1997. [Online]. Available: {citeseer.ist.psu.edu/lehman97metrics.html}

[17] C. Jensen and W. Scacchi, "Role Migration and Advancement Processes in OSSD Projects: A Comparative Case Study," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 364–374.

[18] G. Robles and J. Gonzalez-Barahona, "Contributor Turnover in Libre Software Projects," *Open Source Systems*, pp. 273–286, 2006. [Online]. Available: {http://dx.doi.org/10.1007/0-387-34226-5_28}

[19] G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz, "Evolution of the core team of developers in libre software projects," in *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, May 2009, pp. 167–170.

[20] J.-M. Dalle, M. d. Besten, and H. Masmoudi, "Channeling Firefox Developers: Mom and Dad Aren't Happy," in *Open Source Development, Communities and Quality, IFIP 20th World Computer Congress, Working Group 2.3 on Open Source Software, OSS 2008, September 7-10, 2008, Milano, Italy*, B. Russo, E. Damiani, S. A. Hissam, B. Lundell, and G. Succi, Eds., vol. 275. Springer, 2008, pp. 265–271.

[21] H. Masmoudi, M. d. Besten, C. d. Loupy, and J.-M. Dalle, ""Peeling the Onion": The Words and Actions that Distinguish Core from Periphery in Bug Reports and How Core and Periphery Interact Together." in *OSS: Diverse Communities Interacting, 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009, Skövde, Sweden, June 3-6, 2009. Proceedings*, C. Boldyreff, K. Crowston, B. Lundell, and A. I. Wasserman, Eds., vol. 299. Springer, 2009, pp. 284–297.

[22] M. J. Scialdone, N. Li, R. Heckman, and K. Crowston, "Group Maintenance Behaviors of Core and Peripherial Members of Free/Libre Open Source Software Teams," in *OSS: Diverse Communities Interacting, 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009, Skövde, Sweden, June 3-6, 2009. Proceedings*, C. Boldyreff, K. Crowston, B. Lundell, and A. I. Wasserman, Eds., vol. 299. Springer, 2009, pp. 298–309.

[23] A. Capiluppi and M. Michlmayr, "From the Cathedral to the Bazaar: An Empirical Study of the Lifecycle of Volunteer Community Projects," in *Open Source Development, Adoption and Innovation*, J. Feller, B. Fitzgerald, W. Scacchi, and A. Silitti, Eds. Springer, 2007, pp. 31–44.

[24] E. Capra, C. Francalanci, and F. Merlo, "An Empirical Study on the Relationship Between Software Design Quality, Development Effort and Governance in Open Source Projects," *IEEE Transactions on Software Engineering*, vol. 34, no. 6, pp. 765–782, Nov.-Dec. 2008.

[25] K.-J. Stol and M. A. Babar, "Reporting Empirical Research in Open Source Software: The State of Practice," in *OSS: Diverse Communities Interacting, 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009, Skövde, Sweden, June 3-6, 2009. Proceedings*, C. Boldyreff, K. Crowston, B. Lundell, and A. I. Wasserman, Eds., vol. 299. Springer, 2009, pp. 156–169.

[26] V. Basili, G. Caldiera, and D. H. Rombach, "The Goal Question Metric Approach," in *Encyclopedia of Software Engineering*, J. Marciniak, Ed. Wiley, 1994.

[27] R Development Core Team, *R: A Language and Environment for Statistical Computing*, Vienna, Austria, 2009, ISBN 3-900051-07-0. [Online]. Available: {http://www.R-project.org}

[28] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: an Introduction*. Kluver Academic Publishers, 2000.

[29] D. Barbagallo, C. Francalenei, and F. Merlo, "The Impact of Social Networking on Software Design Quality and Development Effort in Open Source Projects," in *ICIS 2008 Proceedings*, 2008. [Online]. Available: {http://aisel.aisnet.org/icis2008/201}

[30] F. P. Brooks.Jr, *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley , April 1995, ch. "Aristocracy, Democracy, and System Design".